SCALO: Scalability-Aware Parallelism Orchestration for Multi-Threaded Workloads

SCALO: Scalability-Aware Parallelism Orchestration for Multi-Threaded Workloads

GIORGIS GEORGAKOUDIS, Queen's University Belfast
HANS VANDIERENDONCK, Queen's University Belfast
PETER THOMAN, University of Innsbruck
BRONIS R. DE SUPINSKI, Queen's University Belfast
THOMAS FAHRINGER, University of Innsbruck
DIMITRIOS S. NIKOLOPOULOS, Queen's University Belfast

[1] Shared memory machines continue to increase in scale by adding more parallelism through additional cores and complex memory hierarchies. Often, executing multiple applications concurrently, dividing among them hardware threads, provides greater efficiency rather than executing a single application with large thread counts. However, contention for shared resources can limit the improvement of concurrent application execution: orchestrating the number of threads used by each application and is essential.

In this paper we contribute SCALO, a solution to orchestrate concurrent application execution to increase throughput. SCALO monitors co-executing applications at runtime to evaluate their scalability. Its optimizing thread allocator analyzes these scalability estimates to adapt the parallelism of each program. Unlike previous approaches, SCALO differs by including dynamic contention effects on scalability and by controlling the parallelism during the execution of parallel regions. Thus, it improves throughput when other state-of-the-art approaches fail and outperforms them by up to 40% when they succeed.

1 INTRODUCTION

Shared memory machines constitute the building blocks (i.e., *nodes*) of supercomputing and datacenter facilities. These nodes continue to increase in scale in terms of cores and hardware threads. A single, parallel program is often limited in how effectively it can use this increasing hardware parallelism. Co-locating jobs, that is co-executing multiple parallel programs, on a node can increase throughput and energy efficiency [3–5, 27]. As the parallelism continues to increase, efficient execution of some workloads will require node sharing [27] and some systems already enable users to share a node by statically partitioning cores between different jobs[2].

Sharing a node between parallel programs to increase throughput is challenging. Programs scale differently on available resources and co-execution can impact their scalability. Scaling depends on program-specific algorithmic limitations, synchronization requirements and the impact of contention on shared resources, such as the last level cache (LLC) and memory. Moreover, the scalability of a given program varies dynamically between parallel regions as well as

[1] New Paper, Not an Extension of a Conference Paper
[2] https://wiki.anl.gov/cnm/HPC/Submitting_and_Managing_Jobs/Advanced_node_selection
http://www.archer.ac.uk/documentation/user-guide/batch.php

This research has been supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through grant agreement EP/M01147X/1 (SERT).

the obvious limit of sequential ones. Optimized throughput requires thread allocations that reflect these aspects of each co-executing program in a workload as well as their interactions.

Quantifying scalability and the impact of contention is complex. Application scalability depends on (1) the varying application's algorithmic scaling properties; (2) the varying impact of the evolving set of co-located jobs. Existing approaches are limited in accuracy and usability. Offline profiling approaches [12, 18] can miss contention effects that arise during co-execution. Moreover, they require extensive and lengthy runs with different configurations to build scalability models. Prior solutions to quantify scalability at runtime [9, 24] sample performance metrics by experimenting with different thread allocations to evaluate scalability. However, this runtime experimentation wastes execution cycles by bundling measurements with execution [9, 24] and can lead to sub-optimal configurations [12] due to inaccurate characterization of contention.

Even if scalability and contention are accurately quantified, the allocation policies and mechanisms to improve throughput through node sharing remain open questions. The degree of parallelism (*DoP*) of a program execution controls its thread allocation and use of shared resources, such as the memory subsystem. Existing solutions combine policies with scalability profiling choices, through machine learning or other empirical models based on offline [12] or runtime [9] profiles. Irrespective of the allocation policy, prior approaches can only change the DoP at the beginning of execution [12], or before parallel region execution [9] or at thread synchronization points [16, 24]. These limitations can lead to missed opportunities to optimize thread allocations.

We present Scalability-Aware Parallelism Orchestration (SCALO), which overcomes many of the limitations of prior solutions for node sharing, and we demonstrate that it significantly improves throughput. Specifically, we make the following contributions:

- A novel, lightweight method to quantify the scalability of co-running programs that reflects contention. Our method builds an accurate speedup model by periodically sampling a set of key performance indicators to quantify algorithmic, synchronization and contention limitations to scalability. Unlike previous approaches, it does so without requiring runtime experimentation that disrupts execution. SCALO uses this model to estimate the speedup of co-running programs.
- A unique methodology for fine-grain control of the DoP during the execution of parallel regions. Our non-intrusive techniques leverage existing scheduling points within the parallel runtime to control the DoP. A new, adaptive schedule for parallel loops constructs augments DoP control points of loop execution transparently or controllably by the programmer. Our methods break the limitation of previous solutions that change DoP only at the start of a parallel region, never during its execution.
- Two new dynamic allocation policies that increase system throughput, one implementing predictive scaling and the other using an instantaneous, efficiency-based heuristic. Both policies are effective for increasing throughput considerably, by up to 40%, compared to statically dividing cores equally among co-running programs. Notably, they increase throughput when other state-of-the-art runtime approaches fail to do so. Our SCALO allocators, which provide similar throughput increases, show that accurate runtime scalability information and fine-grain control of DoP are the key elements that previous solutions lack rather than a sophisticated policy.

We extensively evaluate SCALO on diverse workloads of co-running programs from the BOTS, NAS, Rodinia and PARSEC suites and compare its allocators with two state-of-the-art solutions: (1) SCAF [8, 9], which is another scalability-aware solution; and (2) an optimizing thread allocator based on a priori collected, offline profiling information. We show that SCALO allocators improve throughput significantly compared to SCAF. For workloads with dynamically

varying scalability, they provide significant improvements compared to the allocator with exhaustive offline profiling and perform similarly for workloads characterized accurately by the offline information.

The rest of the paper is organized as follows: Section 2 presents our method to characterize scalability and contention. Section 3 presents the parallelism orchestrator and discusses adaptivity in the OpenMP runtime. Section 4 presents our performance evaluation. Section 5 discusses related work and Section 6 concludes the paper.

2 SCALABILITY AND SPEEDUP AWARENESS

SCALO provides a significant advance from prior work by accurately and inexpensively modeling scalability and contention at runtime. This section formally defines *scalability* and *speedup* and then presents our model and its implementation of them. A program executes through sequential and parallel regions. Scalability is a property of its parallel regions, quantifies the improvement in their performance as the number of allocated threads, n, increases.

In practice, several factors limit scalability. Synchronization may be required to ensure correctness while the programming model and its implementation may imply other runtime overheads. Machine-level bottlenecks, which may arise from shared resources such as the last-level cache (LLC) or main memory, can reduce scalability within a program and due to interference from co-runners. Typically, single-threaded execution time serves as a reference to quantify scalability under multi-threaded execution. Formally, the execution time of a parallel region p is:

$$T_p(n) = \max t_i,\ i = 1\ \ldots\ n \tag{1}$$

where t_i is the time that thread i executed during that region. If the load on threads is well balanced then each thread executes approximately the same time. Our goal is to quantify scalability without runtime experiments or any other disruption to execution. We contribute a methodology to derive single-threaded execution time $T_p(1)$ by monitoring parallel execution.

We assume programs parallelized by annotating their sequential algorithm to create an equivalent parallel one, using an OpenMP-like programming model. Thus, conceptually, the execution time and number of individual instructions in a multi-threaded execution is the same as the single-threaded execution. However, multi-threaded execution includes additional instructions for synchronization and its time contains extra idle cycles or *stalls*. Stalls due to synchronization and the overhead of the parallel runtime are readily attributed to multi-threaded execution. However, stalls from accessing shared machine resources, such as the LLC and memory, are present both in single-threaded and multi-threaded execution. Some are unavoidable due to poor program locality, while others manifest because of contention, resulting in limited scalability. Thus, we break down the execution time of a thread i as:

$$t_i = t_{i,exe} + t_{i,syn} + t_{i,rt} + t_{i,cont} \tag{2}$$

where $t_{i,exe}$ is the computation time of thread i including any locality stalls, while $t_{i,syn}, t_{i,rt}, t_{i,cont}$ represent time due to synchronization, runtime overhead and time contention stalls. The execution time of the equivalent single-threaded execution is:

$$T_p(1) = \sum_i t_{i,exe},\ i = 1\ \ldots\ n \tag{3}$$

Combining Eqs. 1 and 3, we formally define the speedup of parallel region p as:

$$SP_p(n) = \frac{T_p(1)}{T_p(n)} = \frac{\sum_i t_{i,exe}}{\max t_i},\ i = 1\ \ldots\ n \tag{4}$$

Table 1. Profiling data provided by the runtime

Sample	Description
t_s	Time executed sequentially
t_i	Execution time of thread i
$t_{i,rt}$	Runtime-level execution time of thread i
$t_{i,sync}$	Time spent in synchronization for thread i
$Stalls_{LLC}$	Execution stalls accessing LLC due to an LLC hit
$Stalls_{mem}$	Execution stalls accessing main memory due to an LLC miss
$\overline{Hit}_{L2}$	Average hit ratio of the private L2 caches across threads
$\overline{Hit}_{LLC}$	Average hit ratio of the shared LLC cache across threads

The time to execute all serial and parallel regions of a program determines its overall speedup. Sequential regions are those regions of a program that are not parallelized and, thus, are not controlled by a parallel runtime. These regions execute with one thread without any parallel runtime overheads. In practice, their runtime may vary with the number of threads due to locality effects. However, for simplicity, we assume that their execution time is independent of the number of threads. Thus, the speedup of a program, that is, the ratio of its sequential execution time divided by its parallel execution time with a particular thread allocation for each parallel region, is:

$$SP(p_1, \ldots, p_k) = \frac{T_s + \sum_i T_{p_i}(1)}{T_s + \sum_i T_{p_i}(n_i)} = \frac{T_s + \sum_i \left(T_{p_i}(n_i) \cdot SP_{p_i}(n_i) \right)}{T_s + \sum_i T_{p_i}(n_i)}, i = 1 \ \ldots \ k \tag{5}$$

where T_s is the cumulative execution time of sequential regions and $T_{p_i}(n_i)$ is the time spent executing parallel region p_i with n_i threads.

The model is general to characterize any program running on any micro-architecture, although its parameters will vary on the specific implementation and environment. We base our implementation on these equations. Accurately accounting for stalls and contention is non-trivial. We need novel, specialized performance monitors to analyze stalls accurately [13, 15]. For our implementation, we base profiling on instrumenting the parallel runtime and collecting information from performance counters on existing hardware. Based on collected data, SCALO approximates stalls to quantify scalability and speedup.

2.1 Implementation

We discuss our assumptions and then detail our implementation of our model.

Assumption 1: Cores in the system are not over-subscribed. Thus, no contention occurs for cores or core-private cache components. For our implementation platform, an Intel Ivy Bridge architecture, L1 and L2 caches are core-private. However, contention can exist for other shared components of the memory subsystem (the LLC and memory).

Assumption 2: Programs only use synchronization constructs that the runtime API provides. Thus, we only need to monitor the parallel runtime to measure synchronization time. Even if a program uses fine-grain, user-level locks, they must rely on the runtime API under contention. In our implementation, we instrument an OpenMP runtime that is structured through the fork-join model and explicit directives for synchronization. Our implementation could easily be extended to other parallel runtimes.

We must estimate the sequential execution time of a program while profiling multi-threaded execution with the current thread allocation. The sequential execution time is the reference to estimate scalability of this particular thread allocation. These estimates are the input to optimizing thread allocators to extrapolate scalability for any thread allocation, or as efficiency indicators of the current allocation.

To derive single-threaded performance, SCALO periodically samples timing information and hardware performance counters from the instrumented parallel runtime. Table 1 shows the profiling data. It is possible to measure directly stalls attributed to runtime overhead ($t_{i,rt}$) or synchronization ($t_{i,sync}$). However, we must indirectly evaluate contention stalls ($t_{i,cont}$) because hardware counters measure the total stalls at each level of memory hierarchy, without differentiating between locality and contention stalls. Locality stalls are due to memory access patterns of an application, thus they are unavoidable. Contention stalls are due to multi-threading and an input to the speedup model.

In our experimental architecture, the shared memories are L3, which is the LLC, and the main memory. Profiling measures the total number of stalled cycles accessing the LLC ($Stalls_{LLC}$), when an access misses the L2 but hits the LLC, and main memory ($Stalls_{mem}$), when an access misses the LLC to fetch from main memory. For locality information, profiling measures additionally the hit ratios of core-private L2 caches ($\overline{Hit}_{L2}$), preceding LLC, and the LLC cache ($\overline{Hit}_{LLC}$), preceding main memory. For simplicity, we assume hit ratios remain the same across different degrees of parallelism to derive our approximation.

Intuitively, locality stalls from accessing memory at level j are analogous to the miss ratio of the (cache) memory at level $j-1$. Thus, we approximate locality stalls by factoring the total, measured stalls with the miss ratio of the previous memory in the hierarchy. Taking the LLC for example, a program with good L2 locality rarely accesses the LLC, hence LLC stalls mostly attribute to contention. Note that, in this case, contention is either due to (true or false) data sharing within the program or because of sharing the LLC capacity. By contrast, a program with poor L2 locality has LLC stalls, regardless of contention. For the LLC, locality stalls are approximated by factoring the total LLC stalls with the L2 miss ratio to deduct them from the measured total stalls and derive contention stalls:

$$\widetilde{t}_{cont,LLC} = Stalls_{LLC} - Stalls_{LLC} \cdot \overline{Miss}_{L2} = Stalls_{LLC} \cdot \overline{Hit}_{L2} \tag{6}$$

To clarify more on the approximation, consider the extreme case when $\overline{Miss}_{L2} = 1$ (i.e., $\overline{Hit}_{L2} = 0$), all LLC_{stalls} are the result of poor locality, thus they do not count as contention stalls. For the extreme when $\overline{Miss}_{L2} = 0$ (i.e., $\overline{Hit}_{L2} = 1$), there are no LLC accesses, hence also no LLC contention stalls occur. For the usual case when $0 < Miss_{L2} < 1$, our approximation derives contention stalls after deducting unavoidable stalls attributed to locality.

SCALO uses a similar methodology to approximate memory stalls factoring in the program's LLC miss ratio:

$$\widetilde{t}_{cont,MEM} = Stalls_{mem} - Stalls_{mem} \cdot \overline{Miss}_{LLC} = Stalls_{mem} \cdot \overline{Hit}_{LLC} \tag{7}$$

Contention can affect the LLC hit ratio because of conflict misses from sharing the LLC. Since we cannot directly measure this effect, we approximate the uncontended LLC hit ratio with the hit ratio under contention. We expect hardware to evolve to include more capabilities to monitor shared resources, which would support refinements of our implementation. Combined, Eqs. 6 and 7 approximate the contention stalls from sharing the memory subsystem as:

$$\widetilde{t}_{cont} = Stalls_{LLC} \cdot \overline{Hit}_{L2} + Stalls_{mem} \cdot \overline{Hit}_{LLC} \tag{8}$$

Thus, the estimated sequential execution time (from Eqs. 2, 3, 8) is:

$$\widetilde{T}_p(1) = \sum_i t_{i,exe} = \sum_i t_i - \underbrace{\left(\sum_i \left(t_{i,syn} + t_{i,rt} \right) + \widetilde{t}_{cont} \right)}_{\text{Stalls}}, \; i = 1 \; \ldots \; n$$

where n is the number of threads when profiling.

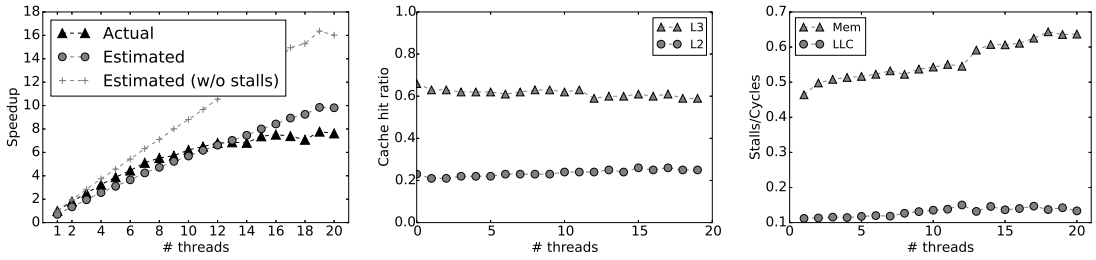
SCALO applies Eqs. 4 and 5 to the measured multi-threaded and estimated sequential execution times to compute the speedup of the sampled period as:

$$\widetilde{SP}_p(n) = \frac{\widetilde{T}_p(1)}{T_p(n)} \text{ and } \widetilde{SP}(n) = \frac{t_s + \widetilde{T}_p(1)}{t_s + T_p(n)}$$

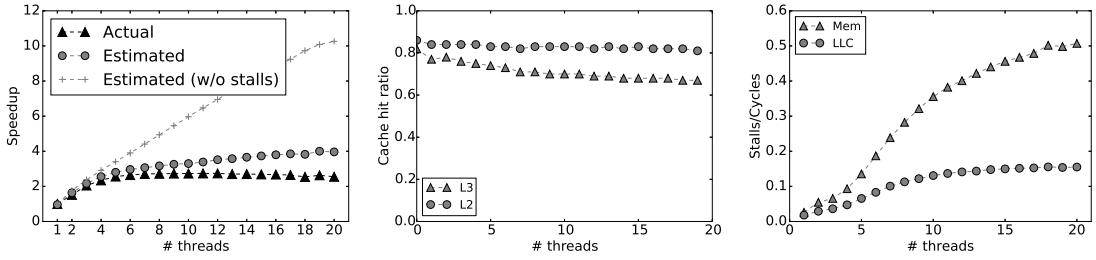
Discussing the portability of the approximation of contention stalls, it is general to apply on different cache and sharing levels. Our methodology approximates locality stalls to subtract from the measured, total stalls and derive contention stalls. The following formula generalizes our method for any shared memory hierarchy:

$$\widetilde{t}_{cont} = \sum_j \left(Stalls_{M_j} - Stalls_{M_j} \cdot \overline{Miss}_{M_{j-1}} \right) = \sum_j \left(Stalls_{M_j} \cdot \overline{Hit}_{M_{j-1}} \right) \tag{9}$$

It sums up contention stalls for all shared memories, where M_j is the shared memory at level j, susceptible to contention. About implementation, SCALO requires hardware counters to measure the cache hit ratio at various levels as well as the number of execution stalls related to a pending miss in any cache in the memory hierarchy. Most current processors include hardware counters that provide these capabilities.

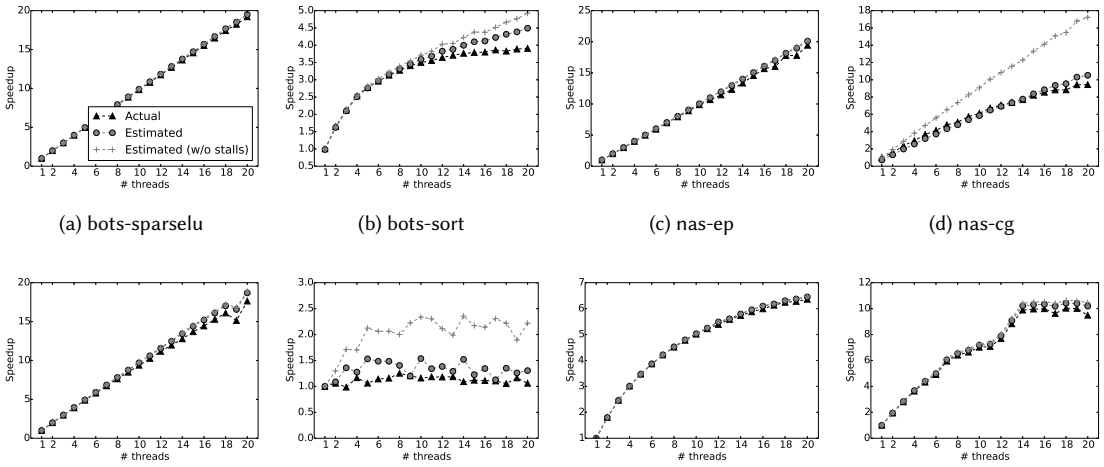


(a) bots-fft



(b) nas-mg

Fig. 1. Speedup and stalls for programs *fft* and *mg*



(a) bots-sparselu (b) bots-sort (c) nas-ep (d) nas-cg

(e) rodinia3-particlefilter (f) rodinia3-backprop (g) parsec3-blackscholes (h) parsec3-fluidanimate

Fig. 2. Speedup estimation for the rest of the programs

2.2 Single program evaluation

We compare actual speedup using different thread counts to the speedup estimates by executing each program individually. This experiment tests the accuracy of the speedup estimation, including the approximation of stalls because of inter-thread contention, for different degrees of parallelism. The same methodology applies to co-running workloads to approximate stalls both from contention due to the program's own threads and interference from co-runners.

Fig. 1 shows the actual speedup, the estimated speedup deducting stalls and the estimated speedup without subtracting memory contention stalls for *fft* and *mg*. The figure also shows graphs on the private L2 and shared LLC cache hit ratios and the ratio of stalls over the total execution time. *fft* is memory bound, irrespective of the number of threads, due to poor locality. Increasing the number of threads has little effect on the LLC cache hit ratio, indicating that contention does not impact LLC effectiveness. However, memory stalls increase because of contention for memory bandwidth. By contrast, *mg* becomes increasingly more memory bound as the DoP increases, by saturating the memory bandwidth as indicated by the sharp rise in memory stalls.

Fig. 2 shows the speedup estimation graphs for the rest of the programs. Across all programs, the average error of speedup estimation is less than 18%. The correction on the estimation by subtracting memory stalls is more important for programs which become increasingly memory bound through scaling, such as *fft*, *mg* and *cg*. For the rest of the programs, the model is accurate by factoring in sequential execution, synchronization and runtime overhead.

Deducting stalls makes the speedup estimation more accurate. Although our platform does not provide hardware performance counters to measure contention stalls directly, our approximation is accurate enough to follow the trend of actual speedup, which is sufficient for evaluating scalability among co-runners. We do not need absolute speedup estimation to guide parallelism orchestration. We apply our method to factor in contention from co-running programs in the workload and demonstrate its effectiveness to guide the SCALO allocators in Section 4.

3 PARALLELISM ORCHESTRATION

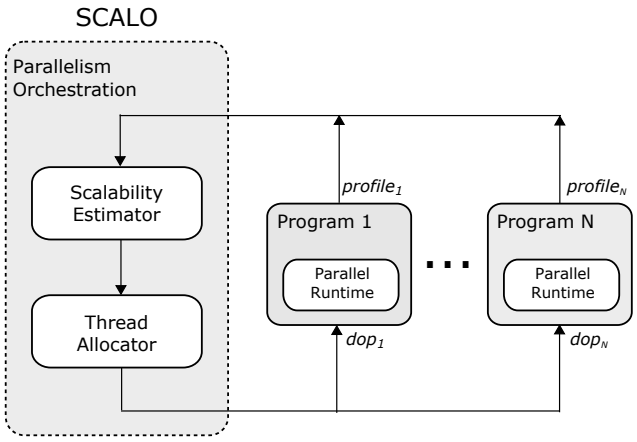


Fig. 3. Overview of SCALO

SCALO runs as a system daemon to have a global view and control over co-executing programs. It can be deployed in a single-user or multi-user setting to optimize compute node throughput when multiple jobs are co-located on it. Fig. 3 shows a diagram of SCALO's functional overview. In detail, SCALO intercepts starting and finishing points in programs and periodically samples profiling data from co-executing programs, by querying their instrumented parallel runtime instances. The scalability estimator derives estimates using timing and performance counter data samples, as described in Section 2. These estimates are the input to the thread allocator module for further analysis. A thread allocator decides the degree of parallelism for each program, by targeting to optimize a system metric of choice. SCALO communicates those thread allocations to the parallel runtimes. SCALO allocators optimize system throughput, in line with other state-of-the-art solutions, formally defined as [14]:

$$STP = \sum_{i=1}^{N} \frac{T_i(1)}{T_{i,co}} = \sum_{i=1}^{N} SP_{i,co} \tag{10}$$

where $T_i(1)$ is the execution time of a program running alone with a single thread, while $T_{i,co}$ is its execution time when co-running under some thread allocation regime. Next, we present the SCALO thread allocators which differ in their approach to scalability modeling and thread allocation.

3.1 Logarithmic Model Allocator

The LOGarithmic Model allocator (LOGM) assumes that the speedup of a parallel region in the program follows a logarithmic curve dependent on the number of threads. This assumption implies that contention effects saturate shared resources and program scaling halts, but without exhibiting any destructive, retrograde effects. LOGM recalculates the logarithmic speedup function with each new profiling sample. Specifically, it fits the latest speedup estimate from the current thread allocation to a logarithmic function using a least-squares method. This function is of the form:

$$SP_p(n) = a \cdot log(n) + b$$

where n is the number of threads. LOGM fits two data points: the basic observation that $SP_p(1) = 1$, thus $b = 1$, and the estimated value $SP_p(n_j) = \frac{\widetilde{T}_p(1)}{T_p(n_j)}$. Note $T_p(n_j)$ is the measured execution time with thread allocation n_j while $\widetilde{T}_p(1)$ is the estimated single-threaded execution time deducting stalls as described in the previous section. LOGM extrapolates speedup by including sequential and parallel execution timing information as in Eq. 5:

$$\widetilde{SP}(n) = \frac{t_s + T_p(n_j) \cdot \widetilde{SP}_p(n)}{t_s + T_p(n_j)} = (1 - f) + f \cdot (a \cdot log(n) + 1) = f \cdot a \cdot log(n), f = \frac{T_p(n_j)}{t_s + T_p(n_j)} \tag{11}$$

LOGM solves an optimization problem to compute the DoP for each program. Specifically, it aims at maximizing the system throughput, as defined in Eq. 10, using the extrapolated speedup from Eq. 11. We set two constraints to limit the solutions of the optimization problem: processor cores must be fully subscribed to ensure full system utilization, and each program must have at least a DoP of 1 to guarantee all co-runners make progress. In formal terms, the maximization problem for N co-executing programs is:

$$\begin{aligned} &\underset{n_1,\ldots,n_N}{\text{maximize}} && \sum_i \widetilde{SP}_i(n_i) \; \forall \; i = 1, \ldots, N \\ &\text{subject to} && \sum_i n_i = C \;\wedge\; n_i \geq 1 \end{aligned}$$

where n_i is the DoP of program i and C is the total number of available processor cores in the system.

We show an example of this approach by assuming two co-executing programs. The extrapolated speedup functions of those programs for Eq. 11 are:

$$\widetilde{SP}_1(n) = f_1 \cdot a_1 \cdot log(n)$$
$$\widetilde{SP}_2(C - n) = f_2 \cdot a_2 \cdot log(C - n)$$

Note that n is the number of threads allocated on the first program, hence the second program has $C - n$ threads to ensure full utilization. Differentiating by n and setting the result equal to zero finds the solution which maximizes the optimization objective:

$$n = \frac{f_1 \cdot a_1 \cdot C}{f_1 \cdot a_1 + f_2 \cdot a_2}, dop_1 = n \;\wedge\; dop_2 = C - n$$

After LOGM computes the DoP for each program, SCALO communicates it to the parallel runtime of the program, provided this results in a different thread allocation than the one in effect.

3.2 Speedup Proportional Allocator

The SPeeDup Proportional allocator (SPDP) improves throughput by following a heuristic approach without an explicit prediction model for scaling. Instead, it calculates instantaneous speedup for each program, using monitoring information. The heuristic that drives the thread allocation of SPDP is that throughput is proportional to speedup, hence SPDP sets the degree of parallelism of each program proportionally to that, as evaluated during co-execution.

Specifically, SPDP applies Eq. 5, updating it after each profiling period with the timing values and the estimated scalability for the specific thread allocation during sampling. SPDP uses this calculated speedup without any extrapolation on scalability. Putting it in formal terms, SPDP computes the DoP for a program i in a workload of N co-executing

programs as:

$$dop_i = \frac{\widetilde{SP_i}}{\sum_j \widetilde{SP_j}} \cdot C \; \forall \; i, j = 1, \ldots, N$$

Note that the calculated *dop* values are rounded to integers. Also, SPDP fully subscribes cores in the system and allocates at least one thread to each program to ensure forward progress.

3.3 Implementation

SCALO is implemented as a daemon process, running in the background. The SCALO daemon communicates with the parallel runtimes through a messaging protocol implemented over shared memory. When a program begins execution, its parallel runtime registers itself with SCALO. SCALO bootstraps thread allocation by equipartitioning cores. As soon as SCALO detects a co-running workload, it starts to periodically sample profiling data of co-runners by probing their instrumented parallel runtimes. Sampling is synchronous, in the sense that SCALO collects samples of all co-executing programs before invoking the scalability estimator, hence it has a complete view of execution. The thread allocator takes input from the scalability estimator to decide an optimizing allocation. SCALO communicates any allocation updates to each parallel runtime for setting the DoP. When a program finishes execution, its parallel runtime unregisters from the daemon to signal exiting the workload. In response to a program exiting, the SCALO daemon invokes again the thread allocator to adjust the DoP for the remaining programs, if there are any.

Next, we present our extensions to the OpenMP runtime, chosen because of its popularity, to make it compatible with SCALO. Our changes can be used as a strategy to render other parallel runtimes SCALO-compatible too.

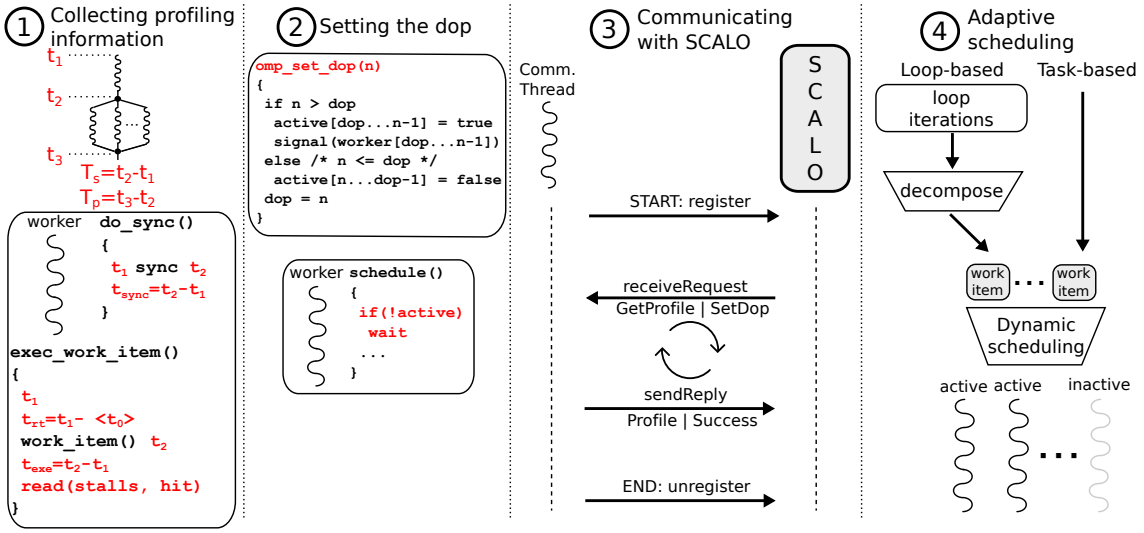


Fig. 4. Extensions to the OpenMP parallel runtime

3.3.1 Implementing Instrumentation and Adaptivity in the OpenMP Runtime. For the implementation we use the Insieme OpenMP compiler and runtime framework [17]. Insieme is OpenMP v3.0 compliant, supporting both loop-based and task-based parallel programs. As for the Insieme runtime [26], it is a high performance OpenMP runtime which is

on par with other widely used runtime implementations, such as GNU OpenMP and Intel OpenMP. We extend the runtime for profiling instrumentation and adaptivity. Fig. 4 summarizes those modifications which we discuss next in detail.

(1) *Collecting profiling information.* The first extension is to instrument the OpenMP runtime for collecting profiling information. The parallel runtime measures timing using timestamps to mark the start and end of execution of a region of interest and calculate its duration. Specifically, profiling measures the duration of sequential and parallel regions, as well as time spent executing runtime, synchronization or program-level code. For collecting hardware counters, we instrument each OpenMP worker thread using PAPI [6] to measure characteristics of program-level code, including stalls and cache hit ratios as shown in Table 1.

(2) *Setting the degree of parallelism.* The second extension is implementing the mechanism to set dynamically the DoP. We extend the runtime interface with a function, namely `omp_set_dop`, which takes as an argument the number of active worker threads to set. Also, we declare an additional, internal control variable to store the DoP value as part of the runtime state, initialized to the total number of threads available to the runtime. Invoking `omp_set_dop` activates or deactivates workers to increase or decrease the number of active threads respectively. Implementation wise, each worker thread has an activity flag and at each scheduling point , a worker checks atomically its private activity flag before starting the execution of a work item. It suspends (blocks on a condition) if it has been tagged inactive. When increasing the DoP, `omp_set_dop` resumes previously blocked worker threads and sets their activity flag. Calling `omp_set_dop` blocks the caller until changing the DoP takes effect.

(3) *Communicating with the SCALO daemon.* For the third extension, the parallel runtime creates a dedicated management thread to communicate with the SCALO daemon. At initialization, the management thread registers the program with SCALO and blocks waiting to receive requests from the daemon. SCALO performs two types of requests: (1) a periodic request for profiling data and (2) an on-demand request to set the DoP for the program. Upon receiving a request for profiling data, the management thread samples timing and hardware counters, then resets them atomically and replies back their values to SCALO. Upon receiving a request to set the DoP, it calls the runtime interface function `omp_set_dop`, waits until it completes and returns a success message to the daemon.

(4) *Enabling fine-grain control of the degree of parallelism.* Superseding other approaches, SCALO changes the DoP during the execution of parallel regions to be more adaptive. There are two prerequisites to have fine-grain control. The first is to have a sufficient number of scheduling points to change the DoP and the second is a dynamic scheduling regime to execute work items on any number of active workers. Task-based programs by design satisfy both those requirements. However, this is not the case for loop-parallel programs, because of loop chunking and scheduling semantics.

The fourth and final extension is to enhance adaptivity for loop-based parallel regions. In OpenMP, a parallel loop can be statically scheduled, equally dividing iterations among worker threads. Although this obviates runtime overhead for chunking and scheduling the loop, it forbids any adaptivity. First of all, the decomposition of loop iterations to work items is too coarse grain to create sufficient scheduling points: a worker cannot be rendered inactive executing a chunk of such a loop. Second, static scheduling of loop iterations may happen at compile time. Notably, this is the default of the OpenMP pass in GCC, thus forbidding any runtime control.

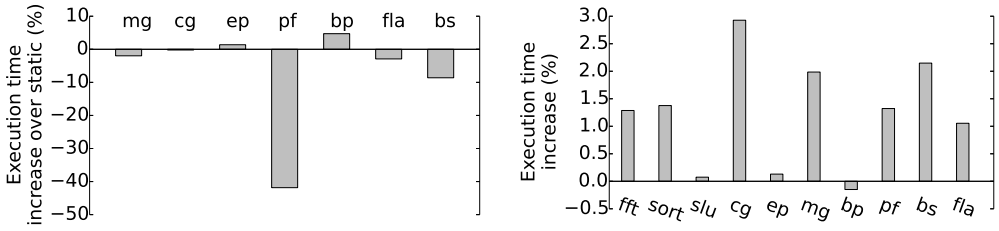
To tackle these problems, we extend the schedule clause of parallel loops in the OpenMP compiler to have an *adaptive* scheduling type, with an optional decomposition factor (df) argument. The semantics for execution are: (1) the runtime decomposes the loop to work items of granularity $\frac{iterations}{df \cdot workers}$ and (2) work items are scheduled dynamically.

If the decomposition factor is not specified, the runtime chooses a default value, set to 8× in our implementation. The OpenMP specification leaves the default loop scheduling as implementation-defined when no schedule clause is explicitly given. Thus, we set adaptive scheduling as the default one.

The methodology of SCALO is applicable to any OpenMP runtime. A SCALO-enabled OpenMP runtime is usable as a drop-in solution with any binary that conforms to the OpenMP API of that runtime. Adaptivity depends on the number of scheduling points. A general limitation for any runtime solution exists when the compiler chunks a statically scheduled loop at compile time, without involving the runtime. Despite that, a SCALO-enabled runtime is still usable as a drop-in solution for the remaining parallel regions. When statically scheduled loops go through the runtime, SCALO applies adaptive scheduling with a default decomposition factor to create extra scheduling points. Nevertheless, we present our extensions to the compiler for explicit adaptive scheduling of parallel loops to let programmers use it if they wish. Also, SCALO is compatible with programs using nested parallelism, since it profiles threads and controls the DoP for each individual parallel region.

3.3.2 *Discussion.*

Using adaptive scheduling by default. Notably, adaptive scheduling can readily override any other type of scheduling but the static one needs special handling. In particular, static scheduling of loops conveys extra semantics at the programming model[3]. This may impose a strict ordering in the execution of loop iterations across work items which must be observed by the runtime if correctness depends on this, such as when the programmer specifies a `nowait` clause. For that, we override static with adaptive scheduling only for those statically scheduled loops for which strict ordering is not required. Interestingly, the latest OpenMP v4.5 specification includes a `taskloop` directive which decomposes a parallel loop to dynamically scheduled OpenMP tasks. It includes a *grain-size* parameter, similar to the decomposition factor, to give a hint to the runtime for slicing loops to tasks.



(a) Comparing adaptive vs. static scheduling

(b) Overhead of SCALO

Fig. 5. Measuring adaptive scheduling and SCALO overhead

Next, we evaluate the performance of adaptive scheduling, with a default decomposition factor of 8×, comparing it with static scheduling. For this comparison, we use loop-based programs taken from NAS, Rodinia and PARSEC benchmark suites and execute them in single-program workloads, fully subscribing cores in the system. Fig. 5(a) shows the results. Adaptive scheduling is on par with static scheduling and notably, in one case, the particlefilter program from Rodinia, execution time reduces significantly because dynamic scheduling balances load better across threads.

[3]By the OpenMP specification of static scheduling, loop constructs schedule iterations to the same worker across loop regions, provided those loops have an identical iteration count and are within the same parallel region.

SCALO overhead. To quantify the overhead of SCALO we show the percentage increase of execution time when instrumentation, sampling and the scalability estimator of SCALO are enabled. The sampling period is set to 100 ms and running the scalability estimator includes the overhead of computations for the speedup model, although its output is unused. Nevertheless, changing the DoP does not require global synchronization between threads as each thread reads a private activity flag (see Fig. 4) to resume or suspend execution to comply with the DoP setting. Fig. 5(b) shows results on all programs, including task-based BOTS programs, again running alone and fully subscribing cores in the system. The mean overhead across all programs is about 1.2%. This overhead is readily recoverable by optimizing thread allocation in SCALO, as we show in the evaluation section.

OpenMP threading API. SCALO is compatible with the existing OpenMP threading interfaces. Specifically, a programmer can control the number of threads either via calling the `omp_set_num_threads()` API function or setting the OMP_NUM_THREADS environment variable. For SCALO, this imposes an upper limit to the number of hardware threads when adapting parallelism. That design choice is motivated by the fact that the programmer sets explicitly the number of threads as a hint on the available parallelism, hence possible scalability. Additionally, a call to the API function `omp_get_num_threads()` returns the maximum number of hardware threads available to the application. This is either the total number of hardware threads, despite SCALO adapting the DoP intermittently, or the upper limit on hardware threads set previously by the programmer. That way, programmers that may use this interface to divide work for parallel execution will always provide enough work items to keep all hardware threads busy, whatever is the SCALO setting of the DoP during execution.

Synchronization and scheduling points. SCALO is safe to use on programs with synchronization. SCALO cannot cause deadlock because it changes the number of threads only at preexisting scheduling points of the OpenMP runtime. Any reduction of threads occurs when the OpenMP runtime would otherwise schedule a new task on one of the worker threads in its `schedule()` function (Fig. 4). If SCALO deactivates a thread, this thread is not executing a task so it cannot hold a lock.

Advanced hardware features. In the section we discuss SCALO in relation to advanced hardware features, including hyperthreading, frequency scaling and TurboBoost, and Thread Level Speculation (TLS).

SCALO evaluates contention in the shared LLC and main memory which is the point of contention for physical cores in a hardware system. Hyperthreads are hardware threads running on the same physical core. Besides sharing the LLC and memory, they also share the lower, core-private levels of cache hierarchy, typically L1 and L2 caches, and the execution ports of the physical core pipeline. SCALO could be extended for hyperthreading, without affecting its OpenMP implementation. Parallel programs may execute up to the number of hyperthreads and profiling can track counters for each hyperthread individually. SCALO's orchestrator would need to reflect contention between hyperthreads, which occurs starting in the L1 cache instead of the LLC. The model would also need to include the stalls from unavailable pipeline ports. Nevertheless, hyperthreading has limited use on compute-bound parallel codes which we experiment with, because it leads to unpredictable variability and often degrades performance.

Frequency scaling and TurboBoost technologies target sequential and interactive applications, typically found in desktop workloads. Frequency scaling reduces power consumption by throttling down cores, avoided on parallel programs as it can cause load imbalance. TurboBoost accelerates sequential performance when there is thermal slack. Targeting HPC installations, SCALO assumes a maximum performance configuration in which cores run at maximum frequency and TurboBoost is deactivated because of multi-threaded execution, fully-subscribing processor cores.

Thread Level Speculation (TLS) is an advanced technique to automatically parallelize sequential applications by extracting tasks to execute speculatively on parallel threads. It requires hardware support to safely commit (roll back) speculative tasks when they preserve (violate) the program's sequential semantics. SCALO could include speculative threads in the DoP of the application, by measuring their impact on contention.

4 EVALUATION

4.1 Experiment methodology

Table 2. Programs categorized by speedup, including execution information

	Program	Suite	Input	$T_s(s)$	$T_p(s)$	# parallel regions	$\geq 2\times$	$\geq 4\times$	$\geq 16\times$	$SP(20)$
High	slu (sparselu)	BOTS	-n 100 -m 200	0.32	39.53	2	0	0	0	19.4
	ep	NAS	class C	0.01	30.58	2	0	0	0	19.3
	pf (particlefilter)	Rodinia	-x 128 -y 128 -z 32 -np 2×10^5	0.14	22.30	20	17	16	16	17.7
Medium	fla (fluidanimate)	PARSEC	native	0.26	84.90	5	0	0	5	9.49
	cg	NAS	class C	1.94	26.57	30	21	19	19	9.2
	fft	BOTS	-n 2^{30}	4.24	41.32	4	0	0	0	7.8
	bs (blackscholes)	PARSEC	native	19.19	7.76	1	0	0	1	6.36
Low	sort	BOTS	-n 2^{30}	36.36	11.75	2	0	0	0	3.9
	mg	NAS	class C	13.67	14.32	19	14	11	9	2.5
	bp (backprop)	Rodinia	-n 2^{26}	50.42	11.73	3	2	0	0	1.06

	Program	Suite	Barriers	Critical regions	Single/Master regions	Reductions	Taskwait
High	slu	BOTS	1	0	1	0	2
	ep	NAS	2	1	2	1	0
	pf	Rodinia	11	0	0	2	0
Medium	fla	PARSEC	5	0	0	0	0
	cg	NAS	26	0	14	6	0
	fft	BOTS	2	0	2	0	16
	bs	PARSEC	1	0	0	0	0
Low	sort	BOTS	1	0	1	0	3
	mg	NAS	19	1	11	0	0
	bp	Rodinia	2	0	0	0	0

The node we experiment with is a dual socket Intel Xeon E5-2690v2 system with 20 physical cores. Each socket hosts 10 cores clocked at 3 GHz. Cores run at their maximum frequency using the `performance` governor of Linux. TurboBoost is deactivated because program workloads fully-subscribe the processors. We turn off SMT to avoid hyperthreading interference in our measurements. Moreover, the machine has 64 GB of DDR3 RAM, distributed equally between the two socket NUMA domains. We use the latest stable (inspire1_3) version of the Insieme OpenMP source-to-source compiler to produce the source fed to GCC 4.8.3 for generating native code.

To create co-running program workloads, we use programs from the BOTS [11], NAS [1], Rodinia [7] and PARSEC [2] benchmark suites. NAS benchmarks represent frequently used HPC kernels with regular parallelism using loop-based

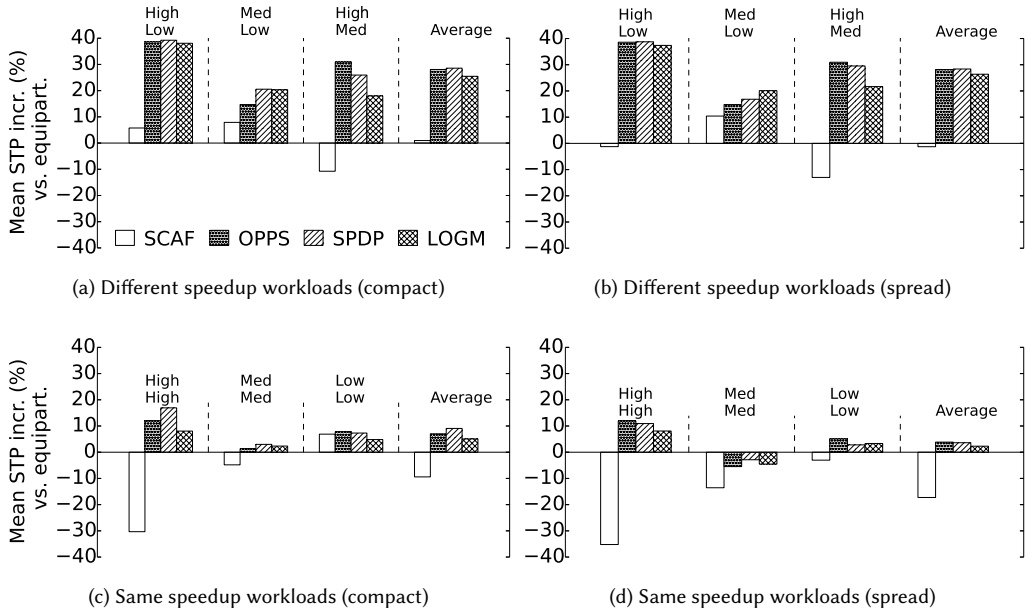
parallelization. BOTS includes programs with irregular parallelism using task-based implementations. Rodinia and PARSEC benchmarks include OpenMP implementations of a variety of applications, besides HPC, targeting multi-processor execution. BOTS programs use task-based parallelization which means they have few parallel regions spawning tasks for execution, whereas NAS, Rodinia and PARSEC programs predominantly use loop-based parallelization and each parallel loop may be a different parallel region. Table 2 presents the benchmark programs, their inputs and additional information characterizing their execution. This includes the total number of parallel regions of the program and their frequency of execution, that is how many parallel regions execute more than 2×, 4× and 16× times. The frequency of parallel region execution is important to understand the limitations of prior work on controlling parallelism and the operation of SCALO. Moreover, the table shows the speedup of each program when executing alone, using all 20 cores in the system.

Referring to table 2, we categorize benchmarks by their speedup of solo execution, when fully subscribing cores in the system. Specifically, those with more than 10× speedup are deemed as high speedup ones, those between 4× and 10× are medium speedup and those with less than 4× are low speedup. We group results by speedup classes to evaluate the optimizing thread allocators.

For experimentation, we use the same experimentation methodology from other state-of-the-art approaches [8, 9]. Co-running workloads include programs both from the same speedup class or different speedup classes to create execution scenarios that test the optimizing allocators. Furthermore, programs start execution at the same time but they may finish at different times. The inputs to programs are chosen so that they have comparable execution times, to maximize co-execution time. A co-running workload has completed when both programs have finished. We record the execution time of each benchmark and compute the system throughput as in Eq. 10. All experiments are repeated for 10 times to report the mean value out of those runs. When error bars are given, they represent 95% confidence intervals.

The baseline for our evaluation is *equipartitioning*, that is equally dividing cores among co-running programs. SCALO implements LOGM and SPDP, as described in section 3. Besides SCALO allocators, we include an implementation of the state-of-the-art allocator SCAF [9], following closely its specification. Moreover, for comparison, we include an allocator using exhaustive offline profiling from solo runs of each program, named OPPS standing for operating points. OPPS takes as input offline collected profiles from solo runs of programs. Those profiles record the execution time of a program for all possible threading configurations, varying the number of threads up to the maximum number of cores and mapping between sockets. OPPS combines offline information of co-runners to choose the thread allocation which maximizes STP. However, OPPS exerts control on parallelism only when a program starts execution, to decide its thread allocation alongside co-runners, or when a program finishes, to adjust the thread allocation on remaining programs. Overall, OPPS represents a design point which has ample offline profiling information from solo execution but lacks the ability to detect contention and scalability variations at runtime to control parallelism.

Concerning SPDP, LOGM and SCAF, we experiment both with *compact* and *spread* mapping of threads to cores. Compact mapping puts threads of the same program to the same socket before crossing to the next socket, if needed. In contrast, spread mapping distributes threads as evenly as possible across sockets. Note that OPPS uses offline profiling information to optimize mapping. Equipartitioning, used as the baseline, always applies compact mapping to isolate programs on different sockets to avoid interference. We leave the automated optimization of thread mapping in addition to dynamically controlling the degree of parallelism as future work.



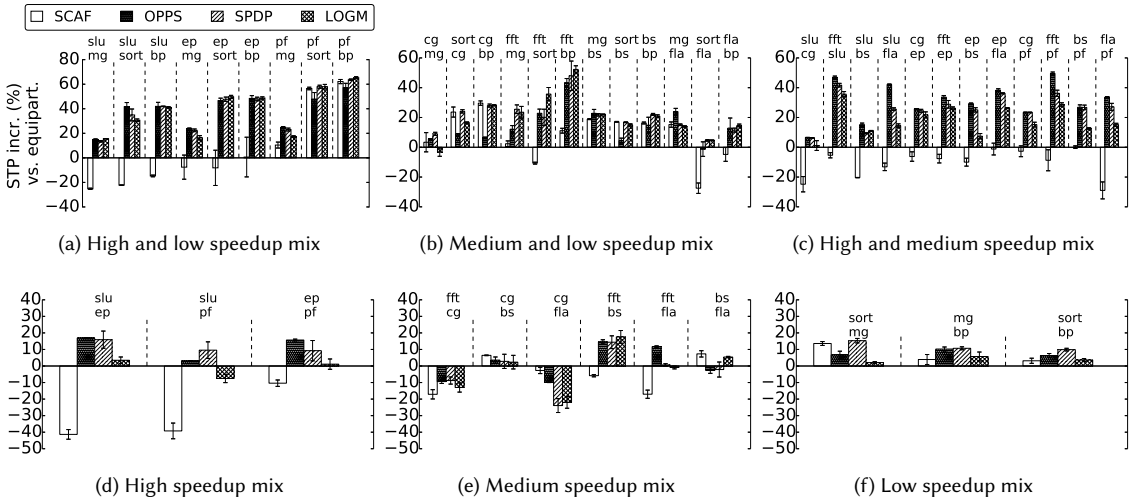
(a) Different speedup workloads (compact)

(b) Different speedup workloads (spread)

(c) Same speedup workloads (compact)

(d) Same speedup workloads (spread)

Fig. 6. Mean throughput increase vs. equipartitioning across 2-program workloads



(a) High and low speedup mix

(b) Medium and low speedup mix

(c) High and medium speedup mix

(d) High speedup mix

(e) Medium speedup mix

(f) Low speedup mix

Fig. 7. Throughput increase vs. equipartitioning for 2-program workloads (compact)

4.2 Results and Discussion

4.2.1 2-program workloads. Fig. 6 shows the mean improvement or deterioration in STP of each allocator per workload group and also summarizes results by averaging across all workloads. Fig. 6(a) – 6(b) present overall results

for different speedup workloads using compact or spread thread mapping and Fig. 6(c) – 6(d) show results for same speedup workloads. Results are similar for compact and spread mapping and Fig. 7 shows the STP improvement for each individual workload using compact mapping.

For different speedup workloads, the SCALO thread allocators SPDP and LOGM effectively improve system throughput, regardless of the thread mapping. Specifically, SPDP and LOGM improve throughput by an average 30% across all workloads. Interestingly, SPDP and LOGM are on par to OPPS, averaging across workloads, but without requiring exhaustive offline profiling. SCAF achieve the lowest throughput improvement, only about 2.5% on average. This result comes from SCAF limitations: the inaccuracy of runtime experimentation when estimating scalability and the inability to control parallelism during parallel region execution.

Drawing general conclusions, the higher the speedup disparity among co-running programs the more SCALO allocators improve throughput. Among the three workload groups, SPDP and LOGM improve throughput the most for the *high-low* speedup workloads, followed by *high-medium* and *medium-low* workloads. This is because the speedup difference between programs justifies moving away from an equipartitioning regime. Nevertheless, even in workloads with less speedup disparity, adapting thread allocation to scalability variations is beneficial, as we elaborate later.

For workloads of programs of the same speedup class, equally dividing cores between them is usually the best strategy since they benefit about the same from additional cores. SCALO results in a similar division of resources as equipartitioning. Thus, SCALO allocators achieve less than 10% improvement on throughput by adjusting DoP for scalability variations. Notably, SCAF results in a net slowdown of about 10% on average because of failing to adapt parallelism to scalability variations during parallel region execution.

Next, we provide additional data and present typical examples from the experimenting workloads to highlight the differences among the deployed allocators and how these affect workload execution.

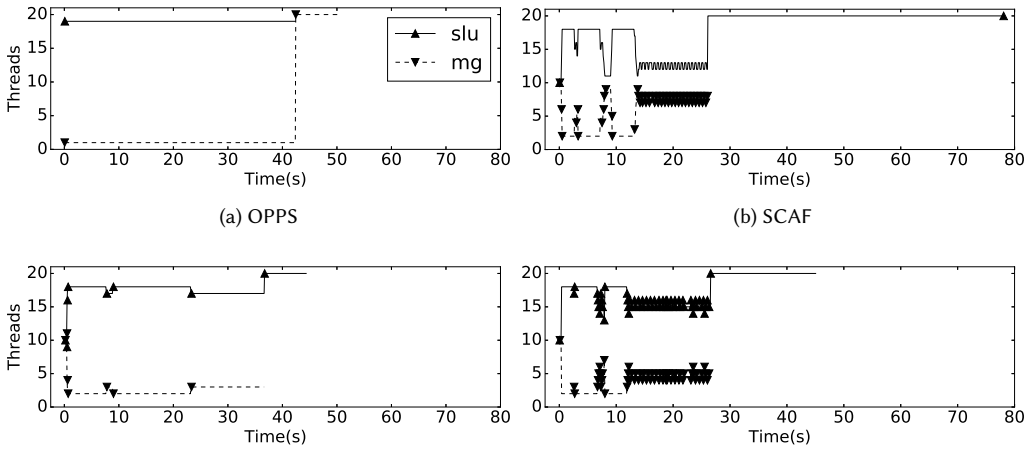


Fig. 8. Thread allocations and STP for the workload *slu-mg*

4.2.2 The importance of controlling parallelism during parallel region execution. Fig. 8(a) – (d) show thread allocations over time for all optimizers when co-executing the programs *slu* and *mg*. Solid lines show what is the optimizing thread allocation each solution computes, while markers show the actual point in time this allocation takes effect.

This is a *high-low* workload: *slu* is the highly-scalable, high speedup program whereas *mg* is the low speedup one due to sequential regions and limited scalability from memory bound execution. For such a workload, *slu* should be allocated more threads than *mg* to optimize throughput. SCALO allocators and OPPS increase STP compared to equipartitioning similarly, by about 15%, but SCAF reduces STP by about 20%.

Notably, SCAF and SCALO allocators have similar decisions for thread allocation. However, SCAF is unable to effectively control parallelism for the *slu* program. In particular, *slu* has a long-running parallel region, within which tasks spawn to perform the computation. SCAF cannot change the parallelism for that region and *slu* runs most of the execution time with the initial allocation of 10 threads, leading to sub-optimal performance. By contrast, SPDP and LOGM adapt parallelism during parallel region execution to effectively improve throughput, being unaffected by program structure. Notably, OPPS performs on par to SCALO allocators by having a priori knowledge of scalability.

Conclusion: changing the degree of parallelism during parallel region execution is necessary to optimize thread allocation, assuming no a priori information on scalability.

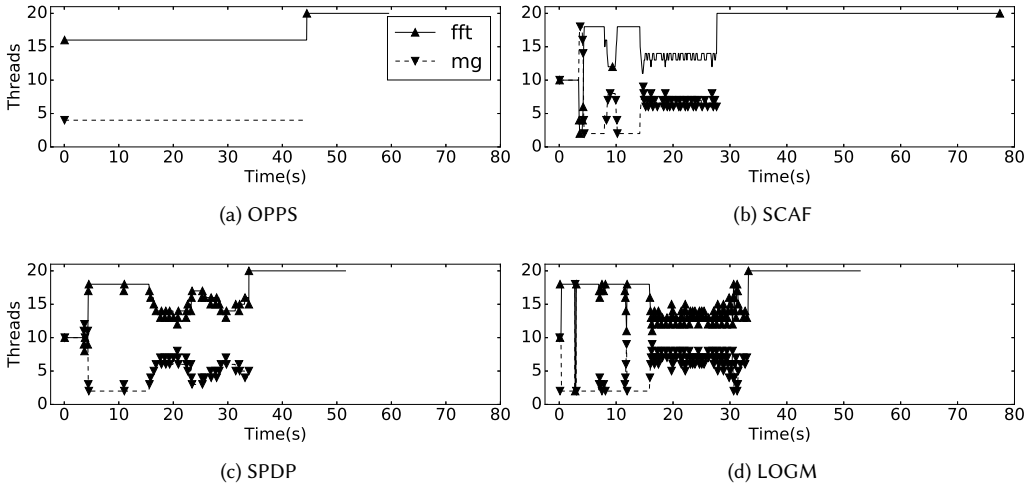


Fig. 9. Thread allocations for the workload *fft-mg*

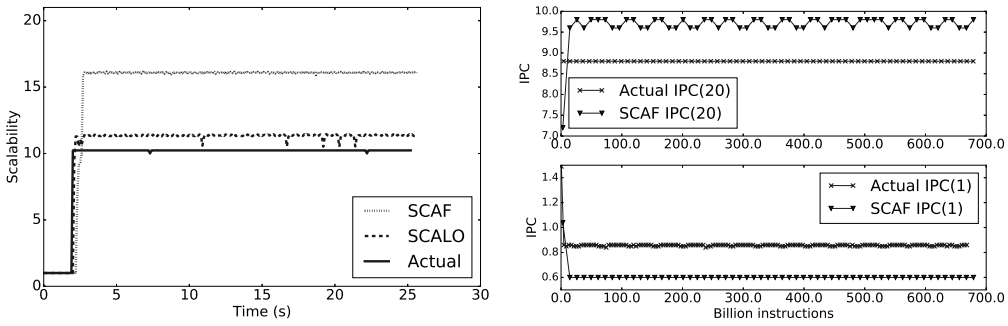
4.2.3 Variations in scalability and fine-grain control of parallelism. Certain workloads experience variations in scalability over time because: (1) co-running programs alternate between sequential and parallel regions and (2) contention affects differently parallel regions of co-runners. This is especially the case for *medium-low* workloads which combine programs with less scaling. Referring to results in Fig. 7(b), OPPS improves throughput consistently less than SCALO allocators on those workloads although it includes offline information on scalability.

As an indicative example, we focus on the *medium-low* workload of co-executing *fft* and *mg*, SPDP improves STP by about 27%, LOGM is on par with 25%, whereas OPPS improves it modestly, about 13%. SCAF again fails to control

parallelism effectively for *fft* due to its few parallel regions and results in an STP improvement of only 3.5%. Fig. 9(a) – (d) show thread allocations over time for the optimizers.

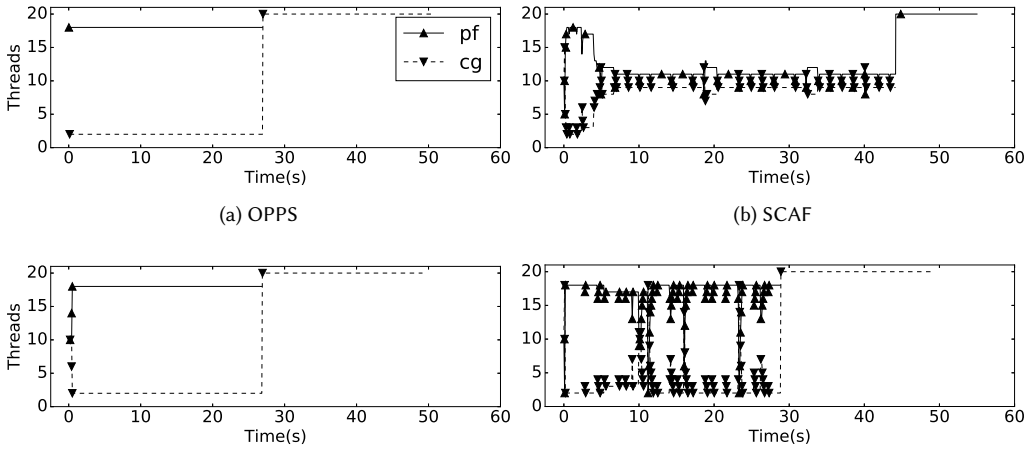
Of all optimizers, OPPS has both the most coarse grain view of scalability and control of parallelism and this limits its scope of optimization. OPPS uses whole program speedup to boost thread allocation on the medium speedup program *fft*. However, SPDP and LOGM identify scalability changes in the workload through online monitoring and adapt parallelism to improve STP considerably more than OPPS.

Conclusion: in workloads with variations in scalability, identifying them at runtime and adapting parallelism improves significantly STP.



(a) Scalability measurement and estimates (b) Measured IPC vs. SCAF

Fig. 10. Scalability estimation and IPC measurements for the program *cg*



(a) OPPS (b) SCAF

(c) SPDP (d) LOGM

Fig. 11. Thread allocations for the workload *pf-cg*

4.2.4 Contention-aware modeling versus runtime experimentation for scalability estimation. As a detailed example, Fig. 10(a) shows actual and estimated scalability for the *cg* program running alone and fully subscribing the 20 cores. This is to compare runtime experimentation, used by SCAF, and the SCALO approach of contention modeling. SCALO results in a much more accurate estimate, being roughly within 10% of the measured scalability while SCAF overestimates it by about 60%.

Going into deeper analysis, Fig. 10(b) shows the actual and SCAF-estimated IPC for single-threaded and multi-threaded execution, focusing on the parallel region of *cg*. Runtime experimentation measures the IPC with $N_{threads} - 1$ and extrapolates this value to the estimated $N_{threads}$. However, this results in overestimation due to ignoring contention effects from extrapolation. Moreover, SCAF measures the single-threaded IPC executing concurrently to the multi-threaded instance. This leads to underestimating the single-threaded IPC because of unaccounted contention effects throttling down single-threaded execution. Since the estimated scalability in SCAF is the ratio of the extrapolated multi-threaded IPC over the single-threaded IPC, these inaccuracies amount to overestimating scalability considerably and the estimates becoming increasingly more inaccurate for programs susceptible to contention.

Fig. 11 indicates the shortcomings on thread allocation when SCAF mischaracterizes scalability. The graph shows thread allocation for all policies, when co-running the programs *pf* and *cg*, a *high-medium* speedup workload. The program *pf* is a compute-bound, highly scalable, high speedup program whereas *cg* is memory bound and thus has limited scalability. All optimizing allocators but SCAF boost thread allocation of *pf*. However, SCAF mischaracterizes the scalability of *cg* and distributes threads equally between *pf* and *cg*, similar to equipartitioning, thus failing to improve system throughput. Notably, SCAF results in a net decrease of STP of about 25% because of the overhead or runtime experimentation.

Conclusion: contention-aware modeling results in more accurate scalability estimates than runtime experimentation, saving also the overhead associated with experimentation.

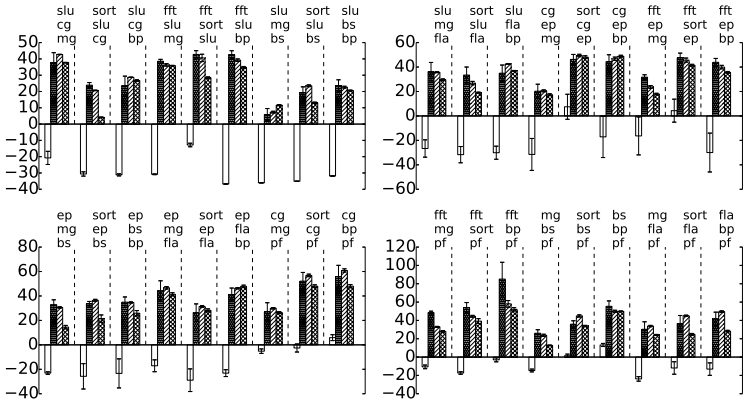


Fig. 12. Throughput increase vs. equipartitioning for 3-program (*high-medium-low*) workloads (compact)

4.2.5 3- and 4-program workloads. For constructing 3- and 4-program workloads, we take all 3-way and 4-way combinations of programs. Fig. 12 shows detailed results only for 3-program workloads of different speedup, due to the large number of experiments. Fig. 13 summarizes results by showing the mean throughput, including the total of possible combinations, for 3- and 4-program workloads. Notably, SCALO allocators maintain good performance,

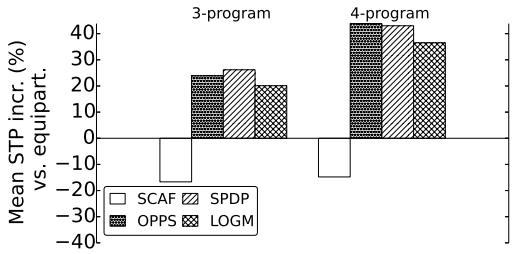


Fig. 13. Mean throughput increase vs. equipartitioning for 3- and 4-program workloads (compact)

increasing throughput by 25% for 3-program workloads and 40% for 4-program workloads, staying on par with the OPPS allocator. SCAF results in an average slowdown of 15%. Increasing the number of programs in the workload exacerbates the overhead from SCAF's runtime experimentation and failing to timely adapt the DoP.

5 RELATED WORK

This section elaborates on the differences between SCALO and prior work. Table 3 compares several related approaches to control DoP with SCALO. We first consider work that optimizes single-program execution but does not consider contention arising from multiprogram workloads.

Table 3. Feature comparison

	Multiple programs	*Runtime profiling*	*Scaling model*	*Contention aware*	*Fine-grain DoP control*	*Avoid runtime experiments*	*Avoid code refactoring*
SCALO	✓	✓	✓	✓	✓	✓	✓
SCAF	✓	✓	✓				✓
Callisto	✓					✓	✓
Varuna	✓	✓	✓				✓
[12]	✓		✓	partially		✓	✓
Tumbler	✓					✓	✓
Lithe						✓	✓
ReSense	✓			✓		✓	✓
Thread Tailor			✓			✓	✓
DoPE			✓		✓	✓	
Parcae	✓	✓	✓		✓		✓
[28]	✓	✓		✓		✓	✓
FDT		✓	✓	✓			✓
Dynamic Teams	✓				✓	✓	✓

5.1 Single-program thread optimizers

Lithe [19] composes parallel runtime libraries that execute in the context of a single program to avoid over-subscription and to improve utilization. Lithe multiplexes user-level threads that execute the libraries on hardware threads. It avoids

over-subscription and intercepts synchronization calls to switch blocked user-level threads to ones that are ready to execute to improve utilization. However, Lithe does not estimate scalability or choose DoPs of the co-executing libraries.

Thread Tailor [18] sets the number of threads of single-program execution to minimize the impact of contention, inter-thread communication and synchronization. It first profiles an instrumented version of the program to record all memory accesses and synchronization overhead. Based on this profile, Thread Tailor guides the DoP of the execution contexts of subsequent runs, using user-level threading, to minimize communication and synchronization overhead. Thread Tailor profiles are input dependent and are prone to overhead and inaccuracies because of instrumentation.

DoPE [21] adjusts DoP based on algorithmic bottlenecks. It provides tools and APIs to enable adaptivity. However, it requires significant development effort: programmers must refactor applications in the DoPE programming model; administrators must set machine-specific optimization targets and constraints; and mechanism developers must implement those constraints through DoP adjustments.

Suleman et al. [25] propose feedback-directed throttling (FDT) to set the DoP for parallel loops. FDT executes the first few iterations of a parallel loop with one thread to evaluate its scalability based on synchronization overhead and memory bandwidth requirements. FDT assumes linear scaling and executes the rest of the loop with the number of threads that minimizes the extrapolated execution time. The training incurs considerable overhead and its contention analysis misses important effects such as sharing the last level cache (LLC).

5.2 Multi-program workload thread allocators

SCAF [8, 9] periodically computes an efficiency metric for each program to guide DoP. SCAF uses runtime experiments that fork single-thread clones of parallel regions and execute them concurrently to the original program running with the remaining threads. It estimates scalability by taking the IPC ratio of multi-threaded over single-threaded execution. However, profiling through forking can have large overhead duplicating process resources. Moreover, contention depends on the number of threads and co-runners; runtime experiments perturbs both, resulting in inaccuracies. Lastly, SCAF can change the DoP only at entry points of parallel regions, foregoing optimization opportunities during the execution of those regions.

Callisto [16] assumes threads in parallel regions are compute bound and allocates hardware threads to user-level contexts using spatial and temporal sharing policies to improve utilization. For temporal sharing, it multiplexes user-level contexts on hardware threads by scheduling out contexts blocked on synchronization. However, Callisto does not model scaling under contention. Instead, it simply assigns cores to programs based on user-defined priorities for spatial sharing.

Varuna [24] estimate scalability based on runtime experiments that periodically measure IPC for different threading configurations of parallel regions. These experiments disrupts execution. Further, Varuna requires co-ordination between co-running programs that can lead to unpredictable delays. Lastly, it can only control DoP at synchronization points, limiting its effectiveness.

Emani et al. [12] observe that measuring performance by varying the number of threads incurs high, unpredictable delays and reduces accuracy. They propose training multiple, machine learning models (*experts*) from solo program runs to predict optimized thread allocations. They deploy a mixture of experts that vary training features and weights, for each program at runtime. They use the expert that best predicts the execution environment before a parallel region starts. The training is time consuming and the quality of the experts depends greatly on choosing appropriate training

weights. Moreover, environment prediction does not model important hardware-level contention effects, such as LLC and memory sharing.

Tumbler [20] changes load balancing in Linux to account for per-thread CPU utilization and migrates threads between cores to equalize utilization. Tumbler increases performance in an over-subscribed system, but has no dynamic scaling model to account for contention and cannot control parallelism.

ReSense [10] changes the thread-to-core mapping for multi-threaded programs to minimize contention. It requires a characterization step, during which a program runs alone to compute a sensitivity score from contention on shared caches and memory. ReSense uses this score at runtime to choose which programs to collocate and which to isolate. It assumes a fixed thread count during execution does not control DoP despite its significant impact on contention.

Parcae [22] is a framework for auto-parallelization of loop-based programs that includes a runtime optimizer for determining the DoP. The Parcae auto-parallelizing compiler constructs a task graph of parallel loops. During execution, the runtime guides an iterative search of thread configurations based on the gradient of a programmer-defined throughput metric. This iterative searching can incur significant overhead, especially for coordinating the search between co-runners, and may lead to a sub-optimal solution trapped in a local minimum.

Zhuravlev et al. [28] adapt the mapping of single-threaded, multiprogram workloads to balance contention across shared cache domains. They use the LLC miss rate as the contention predictor and their online scheduling algorithm migrates programs between shared cache domains to reduce the aggregate miss rate. They do not consider scalability since they only consider single-threaded programs.

Schönherr et al. [23] use dynamic teams that change the number of active threads during parallel regions to reduce over-subscription with co-executing OpenMP programs. They show that equipartitioning hardware threads between co-executing, parallel programs improves performance compared to over-subscription. However, they do not model contention or consider other partitioning strategies. We use equipartitioning as the baseline for our experiments and show that dynamic, scalability-aware partitioning significantly improves system performance.

Earlier solutions have several limitations that reduce their applicability and efficacy. SCALO offers a new solution to avoid those problems. In brief, SCALO contributes lightweight, yet accurate, contention-aware scalability estimation and fine-grain control of DoP during execution to improve system performance.

6 CONCLUSIONS AND FUTURE WORK

In this paper we presented SCALO, a solution to optimize system throughput when co-running parallel programs that share the resources of a node. SCALO monitors the execution of co-running programs at runtime, evaluates contention in vivo and adapts the degree of parallelism to the scalability of each program. SCALO performs better than other state-of-the-art solutions because of two novelties: (1) contention-aware online profiling by using a model based on execution stalls which is less intrusive and more accurate compared to previous offline profiling or runtime experimentation approaches, and (2) controlling the parallelism throughout execution, and critically during the execution of parallel regions, to effectively adapt thread allocation to scalability.

We show two new dynamic allocators possible within SCALO: LOGM and SPDP. LOGM predicts scalability using a logarithmic model to choose those thread allocations for each program which maximize model throughput. Differently, SPDP computes the instantaneous speedup of each program and heuristically allocates threads proportionally to program speedup. Extensive evaluation on workloads that mix programs with different parallelization structures, task or loop-based, and different speedup potential, shows that SCALO allocators, whether predicting or heuristically allocating,

increase throughput significantly. Specifically, SCALO allocators increase throughput on average 30% compared to a baseline of equipartitioning threads among co-runners, while improving throughput more compared to previous state-of-the-art allocators.

As future work, we identify several possibilities for further research. We intend to extend our contention-aware scalability model in two ways. First, to identify contention on other shared resources besides memory, such as accelerators, disks, networking etc., to include a wider class of applications. Second, to improve the model accuracy by proposing new types of performance counters targeted specifically for measuring contention. Furthermore, we are working towards extending SCALO allocators to optimize execution for other metrics besides performance, such as energy consumption. Also, in our scope is to extend parallelism orchestration to include thread mapping techniques for optimizing execution, especially applicable in the presence of hardware heterogeneity, such as NUMA memories and asymmetric processors. Lastly, we work on applying our ideas and techniques on distributed memory parallel execution, for efficiently sharing a cluster of nodes besides sharing a single node.

REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165. DOI: http://dx.doi.org/10.1145/125826.125925

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. DOI: http://dx.doi.org/10.1145/1454115.1454128

[3] Sergey Blagodurov, Alexandra Fedorova, Evgeny Vinnik, Tyler Dwyer, and Fabien Hermenier. 2015. Multi-objective Job Placement in Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 66, 12 pages. DOI: http://dx.doi.org/10.1145/2807591.2807636

[4] Alex D. Breslow, Leo Porter, Ananta Tiwari, Michael Laurenzano, Laura Carrington, Dean M. Tullsen, and Allan E. Snavely. 2016. The case for colocation of high performance computing workloads. 28, 2 (Feb. 2016), 232–251. DOI: http://dx.doi.org/10.1002/cpe.3187

[5] Alex D. Breslow, Ananta Tiwari, Martin Schulz, Laura Carrington, Lingjia Tang, and Jason Mars. 2013. Enabling Fair Pricing on HPC Systems with Node Sharing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 37, 12 pages. DOI: http://dx.doi.org/10.1145/2503210.2503256

[6] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. 2000. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.* 14, 3 (Aug. 2000), 189–204. DOI: http://dx.doi.org/10.1177/109434200001400303

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, Washington, DC, USA, 44–54. DOI: http://dx.doi.org/10.1109/IISWC.2009.5306797

[8] Timothy Creech and Rajeev Barua. 2016. Transparently Space Sharing a Multicore Among Multiple Processes. *ACM Trans. Parallel Comput.* 3, 3, Article 17 (Nov. 2016), 35 pages. DOI: http://dx.doi.org/10.1145/3001910

[9] Timothy Creech, Aparna Kotha, and Rajeev Barua. 2013. Efficient Multiprogramming for Multicores with SCAF. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 334–345. DOI: http://dx.doi.org/10.1145/2540708.2540737

[10] Tanima Dey, Wei Wang, Jack W. Davidson, and Mary Lou Soffa. 2013. ReSense: Mapping Dynamic Workloads of Colocated Multithreaded Applications Using Resource Sensitivity. *ACM Trans. Archit. Code Optim.* 10, 4, Article 41 (Dec. 2013), 25 pages. DOI: http://dx.doi.org/10.1145/2555289.2555298

[11] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *2009 International Conference on Parallel Processing*. 124–131. DOI: http://dx.doi.org/10.1109/ICPP.2009.64

[12] Murali Krishna Emani and Michael O'Boyle. 2015. Celebrating Diversity: A Mixture of Experts Approach for Runtime Mapping in Dynamic Environments. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 499–508. DOI: http://dx.doi.org/10.1145/2737924.2737999

[13] S. Eyerman, K. Du Bois, and L. Eeckhout. 2012. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*. 145–155. DOI: http://dx.doi.org/10.1109/ISPASS.2012.6189221

[14] S. Eyerman and L. Eeckhout. 2008. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro* 28, 3 (May 2008), 42–53. DOI: http://dx.doi.org/10.1109/MM.2008.44

[15] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2006. A Performance Counter Architecture for Computing Accurate CPI

Components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 175–184. DOI: http://dx.doi.org/10.1145/1168857.1168880

[16] Tim Harris, Martin Maas, and Virendra J. Marathe. 2014. Callisto: Co-scheduling Parallel Runtime Systems. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 24, 14 pages. DOI: http://dx.doi.org/10.1145/2592798.2592807

[17] Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. 2012. A Multi-objective Auto-tuning Framework for Parallel Codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 10, 12 pages. http://dl.acm.org/citation.cfm?id=2388996.2389010

[18] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. 2010. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 270–279. DOI: http://dx.doi.org/10.1145/1815961.1815996

[19] Heidi Pan, Benjamin Hindman, and Krste Asanović. 2010. Composing Parallel Software Efficiently with Lithe. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 376–387. DOI: http://dx.doi.org/10.1145/1806596.1806639

[20] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Tumbler: An Effective Load-Balancing Technique for Multi-CPU Multicore Systems. *ACM Trans. Archit. Code Optim.* 12, 4, Article 36 (Nov. 2015), 24 pages. DOI: http://dx.doi.org/10.1145/2827698

[21] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. 2011. Parallelism Orchestration Using DoPE: The Degree of Parallelism Executive. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 26–37. DOI: http://dx.doi.org/10.1145/1993498.1993502

[22] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: A System for Flexible Parallel Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 133–144. DOI: http://dx.doi.org/10.1145/2254064.2254082

[23] J. H. Schonherr, J. Richling, and H. U. Heiss. 2010. Dynamic Teams in OpenMP. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*. 231–237. DOI: http://dx.doi.org/10.1109/SBAC-PAD.2010.36

[24] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2014. Adaptive, Efficient, Parallel Execution of Parallel Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 169–180. DOI: http://dx.doi.org/10.1145/2594291.2594292

[25] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. 2008. Feedback-driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 277–286. DOI: http://dx.doi.org/10.1145/1346281.1346317

[26] Peter Thoman. 2013. Insieme-RS: a compiler-supported parallel runtime system. (2013).

[27] Joseph P. White, Robert L. DeLeon, Thomas R. Furlani, Steven M. Gallo, Matthew D. Jones, Amin Ghadersohi, Cynthia D. Cornelius, Abani K. Patra, James C. Browne, William L. Barth, and John Hammond. 2014. An Analysis of Node Sharing on HPC Clusters Using XDMoD/TACC_Stats. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE '14)*. ACM, New York, NY, USA, Article 31, 8 pages. DOI: http://dx.doi.org/10.1145/2616498.2616533

[28] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 129–142. DOI: http://dx.doi.org/10.1145/1736020.1736036